08 | API：在 WebAssembly MVP 标准下你能做到哪些事?

2020-09-21 于航

WebAssembly入门课 进入课程 >

讲述：于航
时长 21:23 大小 19.59M

你好，我是于航。

在目前与 Wasm 相关的一系列标准中，我们可以将这些标准主要分为两个部分：“Wasm 核心标准（Core Interfaces）”以及“嵌入接口标准（Embedding interfaces）”。

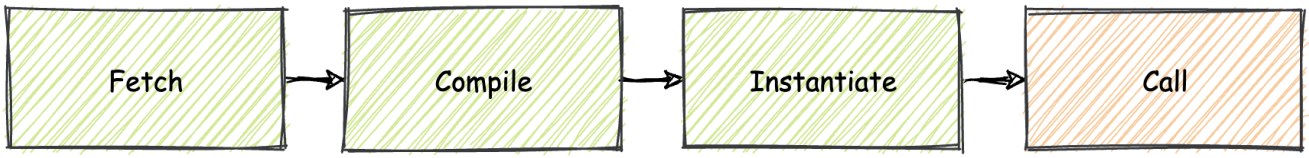
其中，“Wasm 核心标准”主要定义了与 “Wasm 字节码”、“Wasm 模块结构”、“WAT 可读文本格式”以及模块验证与指令执行细节等相关的内容。关于这部分标准中的内容，我在前面几节课中，已经有选择性地为你挑选了部分重点进行解读。

而另一个标准“嵌入接口标准”，则定义了有关 Wasm 在 Web 平台上，在与浏览器进行交互时所需要使用的相关 Web 接口以及 JavaScript 接口。在本节课里，我们将讨论有关

于这些 API 接口的内容。相信在学完本节课后你便会知道，在当前的 MVP 标准下，我们能够使用 Wasm 在 Web 平台上做些什么？哪些又是 Wasm 暂时无法做到的？

Wasm 浏览器加载流程

那在开始真正讲解这些 API 之前，我们先来看一看，一个 Wasm 二进制模块需要经过怎样的流程，才能够最终在 Web 浏览器中被使用。你可以参考一下我画的这张图，这些流程可以被粗略地划分为以下四个阶段。



首先是 “Fetch” 阶段。作为一个客户端 Web 应用，在这个阶段中，我们需要将被使用到的 Wasm 二进制模块，从网络上的某个位置通过 HTTP 请求的方式，加载到浏览器中。

这个 Wasm 二进制模块的加载过程，同我们日常开发的 Web 应用在浏览器中加载 JavaScript 脚本文件等静态资源的过程，没有任何区别。对于 Wasm 模块，你也可以选择将它放置到 CDN 中，或者经由 Service Worker 缓存，以加速资源的下载和后续使用过程。

接下来是 “Compile” 阶段。在这个阶段中，浏览器会将从远程位置获取到的 Wasm 模块二进制代码，编译为可执行的平台相关代码和数据结构。这些代码可以通过 “postMessage()” 方法，在各个 Worker 线程中进行分发，以让 Worker 线程来使用这些模块，进而防止主线程被阻塞。此时，浏览器引擎只是将 Wasm 的字节码编译为平台相关的代码，而这些代码还并没有开始执行。

紧接着便是最为关键的 “Instantiate” 阶段。在这个阶段中，浏览器引擎开始执行在上一步中生成的代码。在前面的几节课中我们曾介绍过，Wasm 模块可以通过定义 “Import Section” 来使用外界宿主环境中的一些资源。

在这一阶段中，浏览器引擎在执行 Wasm 模块对应的代码时，会将那些 Wasm 模块规定需要从外界宿主环境中导入的资源，导入到正在实例化中的模块，以完成最后的实例化过

程。这一阶段完成后，我们便可以得到一个动态的、保存有状态信息的 Wasm 模块实例对象。

最后一步便是 “Call” 。顾名思义，在这一步中，我们便可以直接通过上一阶段生成的动态 Wasm 模块对象，来调用从 Wasm 模块内导出的方法。

接下来，我们将围绕上述流程中的第二步 “Compile 编译” 与第三步 “Instantiate 实例化”，来分别介绍与这两个阶段相关的一些 JavaScript API 与 Web API。

Wasm JavaScript API

模块对象

映入眼帘的第一个问题就是，我们如何在 JavaScript 环境中表示刚刚说过的 “Compile 编译” 与 “Instantiate 实例化” 这两个阶段的 “产物” ？为此，Wasm 在 JavaScript API 标准中为我们提供了如下两个对象与之分别对应：

WebAssembly.Module

WebAssembly.Instance

不仅如此，上面这两个 JavaScript 对象本身也可以被作为类型构造函数使用，以用来直接构造对应类型的对象。也就是说，我们可以通过 “new” 的方式并传入相关参数，来构造这些类型的某个具体对象。比如，可以按照以下方式来生成一个 WebAssembly.Module 对象：

复制代码

```
// "..." 为有效的 Wasm 字节码数据;
bufferSource = new Int8Array([...]);
let module = new WebAssembly.Module(bufferSource);
```

这里的 WebAssembly.Module 构造函数接受一个包含有效 Wasm 二进制字节码的 ArrayBuffer 或者 TypedArray 对象。

WebAssembly.Instance 构造函数的用法与 WebAssembly.Module 类似，只不过是构造函数的参数有所区别。更详细的 API 使用信息，你可以点击这里进行参考。

导入对象

我们曾在之前的课程中介绍过 Wasm 二进制模块内部 “Import Section” 的作用。通过这个 Section，模块便可以在实例化时接收并使用来自宿主环境中的数据。

Web 浏览器作为 Wasm 模块运行时的一个宿主环境，通过 JavaScript 的形式提供了可以被导入到 Wasm 模块中使用的数据类型，这些数据类型包括函数（Function）、全局数据（Global）、线性内存对象（Memory）以及 Table 对象（Table）。其中除“函数”类型外，其他数据类型分别对应着以下由 JavaScript 对象表示的包装类型：

WebAssembly.Global

WebAssembly.Memory

WebAssembly.Table

而对于函数类型，我们可以直接使用 JavaScript 语言中的“函数”来作为代替。

同理，我们也可以通过“直接构造”的方式来创建上述这些 JavaScript 对象。以 “WebAssembly.Memory” 为例，我们可以通过如下方式，来创建一个 WebAssembly.Memory 对象：

复制代码

```
let memory = new WebAssembly.Memory({
  initial:10,
  maximum:100,
});

```

这里我们通过为构造函数传递参数的方式，指定了所生成 WebAssembly.Memory 对象的一些属性。比如该对象所表示的 Wasm 线性内存其初始大小为 10 页，其最大可分配大小为 100 页。

需要注意的是，Wasm 线性内存的大小必须是 “Wasm 页” 大小的整数倍，而一个 “Wasm 页” 的大小在 MVP 标准中被定义为了 “64KiB” （注意和 64 KB 的区别。KiB 为 1024 字节，而 KB 为 1000 字节）。

关于另外的 WebAssembly.Global 与 WebAssembly.Table 这两个类型所对应构造函数的具体使用方式，你可以点击这里进行参考。

错误对象

除了上述我们介绍的几个比较重要的 JavaScript WebAssembly 对象之外，还有另外几个与 “Error” 有关的表示某种错误的 “错误对象” 。这些错误对象用以表示在整个 Wasm 加载、编译、实例化及函数执行流程中，在其各个阶段中所发生的错误。这些错误对象分别是：

WebAssembly.CompileError 表示在 Wasm 模块编译阶段（Compile）发生的错误，比如模块的字节码编码格式错误、魔数不匹配

WebAssembly.LinkError 表示在 Wasm 模块实例化阶段（Instantiate）发生的错误，比如导入到 Wasm 模块实例 Import Section 的内容不正确

WebAssembly.RuntimeError 表示在 Wasm 模块运行时阶段（Call）发生的错误，比如常见的 “除零异常”

上面这些错误对象也都有对应的构造函数，可以用来构造对应的错误对象。（同样，如果有需要，你可以点击这里进入 MDN 网站参考一下）

模块实例化方法

最后一个需要重点介绍的 JavaScript API 主要用来实例化一个 Wasm 模块对象。该方法的原型如下所示：

```
WebAssembly.instantiate(bufferSource, importObject)
```

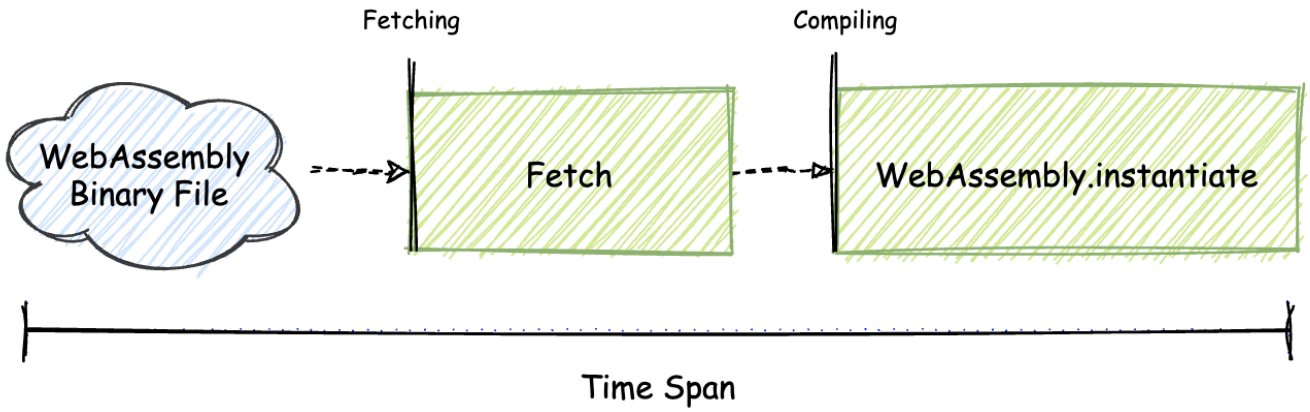
这个方法接受一个包含有效 Wasm 模块二进制字节码的 ArrayBuffer 或 TypedArray 对象，然后返回一个将被解析为 WebAssembly.Module 的 Promise 对象。就像我上面讲的那样，这里返回的 WebAssembly.Module 对象，代表着一个被编译完成的 Wasm 静态模块对象。

整个方法接受两个参数。除第一个参数对应的 ArrayBuffer 或 TypedArray 类型外，第二个参数为一个 JavaScript 对象，在其中包含有需要被导入到 Wasm 模块实例中的数据，这些数据将通过 Wasm 模块的 “Import Section” 被导入到模块实例中使用。

方法在调用完成后会返回一个将被解析为 ResultObject 的 Promise 对象。ResultObject 对象包含有两个字段，分别是 “module” 以及 “instance”。

其中 module 表示一个被编译好的 WebAssembly.Module 静态对象；instance 表示一个已经完成实例化的 WebAssembly.Instance 动态对象。所有从 Wasm 模块中导出的方法，都被“挂载”在这个 ResultObject 对象上。

基于这个方法实现的 Wasm 模块初始化流程如下图所示。你可以看到，整个流程是完全串行的。



需要注意的是，WebAssembly.instantiate 方法还有另外的一个重载形式，也就是其第一个参数类型从含有 Wasm 模块字节码数据的 bufferSource，转变为已经编译好的静态 WebAssembly.Module 对象。这种重载形式通常用于 WebAssembly.Module 对象已经被提前编译好的情况。

模块编译方法

上面讲到的 WebAssembly.instantiate 方法，主要用于从 Wasm 字节码中一次性进行 Wasm 模块的编译和实例化过程，而这通常是我们经常使用的一种形式。当然你也以将编译和实例化两个步骤分开来进行。比如单独对于编译阶段，你可以使用下面这个 JavaScript API：

```
WebAssembly.compile(bufferSource)
```

该方法接收一个含有有效 Wasm 字节码数据的 bufferSource，也就是 ArrayBuffer 或者 TypedArray 对象。返回的 Promise 对象在 Resolve 后，会返回一个编译好的静态 WebAssembly.Module 对象。

Wasm Web API

Wasm 的 JavaScript API 标准，主要定义了一些与 Wasm 相关的类型和操作，这些类型和操作与具体的平台无关。为了能够在最大程度上利用 Web 平台的一些特性，来加速 Wasm 模块对象的编译和实例化过程，Wasm 标准又通过添加 Wasm Web API 的形式，为 Web 平台上的 Wasm 相关操作提供了新的、高性能的编译和实例化接口。

模块流式实例化方法

不同于 JavaScript API 中的 WebAssembly.instantiate 方法，Web API 中定义的“流式接口”可以让我们提前开始对 Wasm 模块进行编译和实例化过程，你也可以称此方式为“流式编译”。比如下面这个 API 便对应着 Wasm 模块的“流式实例化”接口：

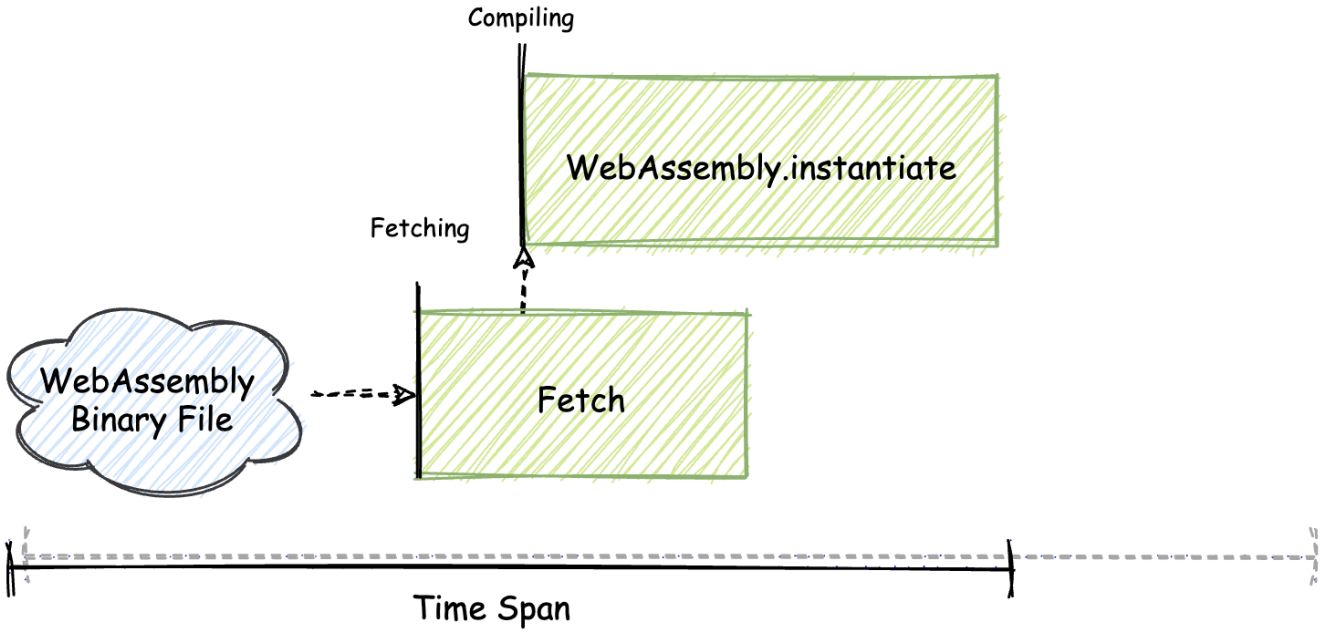
```
WebAssembly.instantiateStreaming(source, importObject)
```

为了能够支持“流式编译”，该方法的第一个参数，将不再需要已经从远程加载好的完整 Wasm 模块二进制数据（bufferSource）。取而代之的，是一个尚未 Resolve 的 Response 对象。

Response 对象（window.fetch 调用后的返回结果）是 Fetch API 的重要组成部分，这个对象代表了某个远程 HTTP 请求的响应数据。而该方法中第二个参数所使用的 Response 对象，则必须代表着对某个位于远程位置上的 Wasm 模块文件的请求响应数据。

通过这种方式，Web 浏览器可以在从远程位置开始加载 Wasm 模块文件数据的同时，也一并启动对 Wasm 模块的编译和初始化工作。相较于上一个 JavaScript API 需要在完全获取 Wasm 模块文件二进制数据后，才能够开始进行编译和实例化流程的方式，流式编译无疑在某种程度上提升了 Web 端运行 Wasm 应用的整体效率。

基于流式编译进行的 Wasm 模块初始化流程如下图所示。可以看到，与之前 API 有所不同的是，Wasm 模块的编译和初始化可以提前开始，而不用再等待模块的远程加载完全结束。因此应用的整体初始化时间也会有所减少。



模块流式编译方法

那么既然存在着模块的“流式实例化方法”，便也存在着“流式编译方法”。如下所示：

```
WebAssembly.compileStreaming(source)
```

该 API 的使用方式与 WebAssembly.instantiateStreaming 类似，第一个参数为 Fetch API 中的 Response 对象。API 调用后返回的 Promise 对象在 Resolve 之后，会返回一个编译好的静态 WebAssembly.Module 对象。

同 Wasm 模块的“流式实例化方法”一样，“流式编译方法”也可以在浏览器加载 Wasm 二进制模块文件的同时，提前开始对模块对象的编译过程。

Wasm 运行时（Runtime）

这里提到的“运行时”呢，主要存在于我们开头流程图中的 “Call” 阶段。在这个阶段中，我们可以调用从 Wasm 模块对象中导出的函数。每一个经过实例化的 Wasm 模块对象，都会在运行时维护自己唯一的“调用栈”。

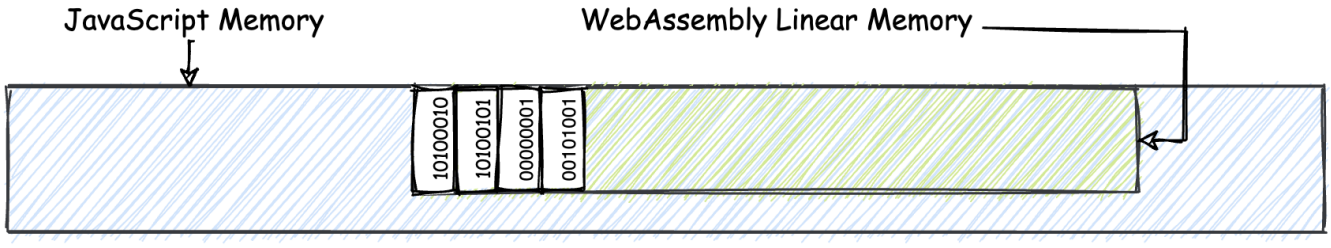
所有模块导出函数的实际调用过程，都会影响着栈容器中存放的数据，这些数据代表着每条 Wasm 指令的执行结果。当然，这些结果也同样可以被作为导出函数的返回值。

调用栈一般是“不透明”的。也就是说，我们无法通过任何 API 或者方法直接接触到栈容器中存放的数据。因此，这也是 Wasm 保证执行安全的众多因素之一。

除了调用栈，每一个实例化的 Wasm 模块对象都有着自己的（在 MVP 下只能有一个）线性内存段。在这个内存段中，以二进制形式存放着 Wasm 模块可以使用的所有数据资源。

这些资源可以是来自于对 Wasm 模块导出方法调用后的结果，即通过 Wasm 模块内的相关指令对线性内存中的数据进行读写操作；也可以是在进行模块实例化时，我们将预先填充好的二进制数据资源以 WebAssembly.Memory 导入对象的形式，提前导入到模块实例中进行使用。

浏览器在为 Wasm 模块对象分配线性内存时，会将这部分内存与 JavaScript 现有的内存区域进行隔离，并单独管理，你可以参考我下面给你画的这张图。在以往的 JavaScript Memory 中，我们可以存放 JavaScript 中的一些数据类型，这些数据同时也可以被相应的 JavaScript / Web API 直接访问。而当数据不再使用时，它们便会被 JavaScript 引擎的 GC 进行垃圾回收。



相反，图中绿色部分的 WebAssembly Memory 则有所不同。这部分内存可以被 Wasm 模块内部诸如 “i32.load” 与 “i32.store” 等指令直接使用，而外部浏览器宿主中的 JavaScript / Web API 则无法直接进行访问。不仅如此，分配在这部分内存区域中的数据，受限于 MVP 中尚无 GC 相关的标准，因此需要 Wasm 模块自行进行清理和回收。

Wasm 的内存访问安全性是众多人关心的一个话题。事实上你并不用担心太多，因为当浏览器在执行 “i32.load” 与 “i32.store” 这些内存访问指令时，会首先检查指令所引用的内存地址偏移，是否超出了 Wasm 模块实例所拥有的内存地址范围。若引用地址不在上图中绿色范围以内，则会终止指令的执行，并抛出相应的异常。这个检查过程我们一般称之为 “Bound Check”。

那么，接下来我们再把目光移到 WebAssembly Memory 身上，来看一看它是如何与“浏览器”这个 Web 宿主环境中的 JavaScript API 进行交互的。

Wasm 内存模型

根据之前课程所讲的内容，我们知道，每一个 Wasm 模块实例都有着自己对应的线性内存段。准确来讲，也就是由 “Memory Section” 和 “Data Section” 共同“描述”的一个线性内存区域。在这个区域中，以二进制形式存放着模块所使用到的各种数据资源。

事实上，每一个 Wasm 实例所能够合法访问的线性内存范围，仅限于我们上面讲到的这一部分内存段。对于宿主环境中的任何变量数据，如果 Wasm 模块实例想要使用，一般可以通过以下两种常见的方式：

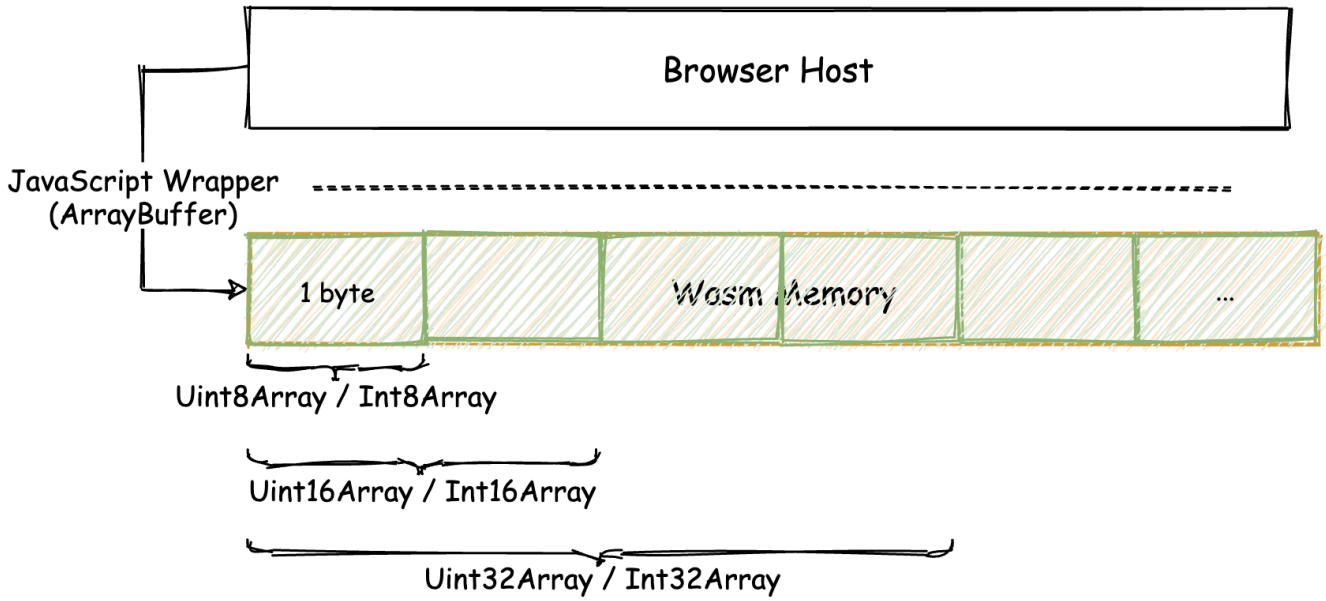
1. 对于简单（字符 \ 数字值等）数据类型，可以选择将其视为全局数据，通过 “Import Section” 导入到模块中使用；
2. 对于复杂数据，需要将其以“字节”的形式，拷贝到模块实例的线性内存段中来使用。

在 Web 浏览器这个宿主环境中，一个内存实例通常可以由 JavaScript 中的 ArrayBuffer 类型来进行表示。ArrayBuffer 中存放的是原始二进制数据，因此在需要读写这段数据时，我们必须指定一个“操作视图（View）”。你可以把“操作视图”理解为，在对这些二进制数据进行读写操作时，数据的“解读方式”。

举个例子，假设我们想要将字符串 “Hello, world!” ，按照逐个字符的方式写入到线性内存段中，那么在进行写操作时，我们如何知道一个字符所应该占用的数据大小呢？

根据实际需要，一个字符可能会占用 1 个字节到多个字节不等的大小。而这个“占用大小”便是我们之前提到的数据“解读方式”。在 JavaScript 中，我们可以使用 TypedArray 以某个具体类型作为视图，来操作 ArrayBuffer 中的数据。

你可以通过下面这张图，来理解一下我们刚刚说的 Wasm 模块线性内存与 Web 浏览器宿主环境，或者说与 JavaScript 之间的互操作关系。



当我们拥有了填充好数据的 ArrayBuffer 或 TypedArray 对象时，便可以构造自己的 WebAssembly.Memory 导入对象。然后在 Wasm 模块进行实例化时，将该对象导入到模块中，来作为模块实例的线性内存段进行使用。

局限性

一切看起来好像都还不错，但我们现在再来回味一下 MVP 的全称。MVP 全称为“Minimum Viable Product”，翻译过来是“最小可用产品”。那既然是“最小可用”，当然也就意味着它还有很多的不足。我给你总结了一下，目前可以观测到的“局限性”主要集中在以下几个方面：

无法直接引用 DOM

在 MVP 标准下，我们无法直接在 Wasm 二进制模块内引用外部宿主环境中的“不透明”（即数据内部的实际结构和组成方式未知）数据类型，比如 DOM 元素。

因此目前通常的一种间接实现方式是使用 JavaScript 函数来封装相应的 DOM 操作逻辑，然后将该函数作为导入对象，导入到模块中，由模块在特定时机再进行间接调用来使用。但相对来说，这种借助 JavaScript 的间接调用方式，在某种程度上还是会产生无法弥补的性能损耗。

复杂数据类型需要进行编解码

还是类似的问题，对于除“数字值”以外的“透明”数据类型（比如字符串、字符），当我们想要将它们传递到 Wasm 模块中进行使用时，需要首先对这些数据进行编码（比如 UTF-8）。然后再将编码后的结果以二进制数据的形式存放到 Wasm 的线性内存段中。模块内部指令在实际使用时，再将这些数据进行解码。

因此我们说，就目前 MVP 标准而言，Wasm 模块的线性内存段是与外部宿主环境进行直接信息交换的最重要“场所”。

总结

好了，讲到这，今天的内容也就基本结束了。最后我来给你总结一下。

在本节课中，我们主要讲解了 Wasm MVP 相关标准中的 JavaScript API 与 Web API。借助这些 API，我们可以在 Web 平台上通过 JavaScript 代码来与 Wasm 模块进行一系列的交互。

我们可以用一句话来总结目前 Wasm MVP 标准在 Web 浏览器上的能力：**凡是能够使用 Wasm 来实现的功能，现阶段都可以通过 JavaScript 来实现；而能够使用 JavaScript 来实现的功能，其中部分还无法直接通过 Wasm 实现（比如调用 Web API）**。

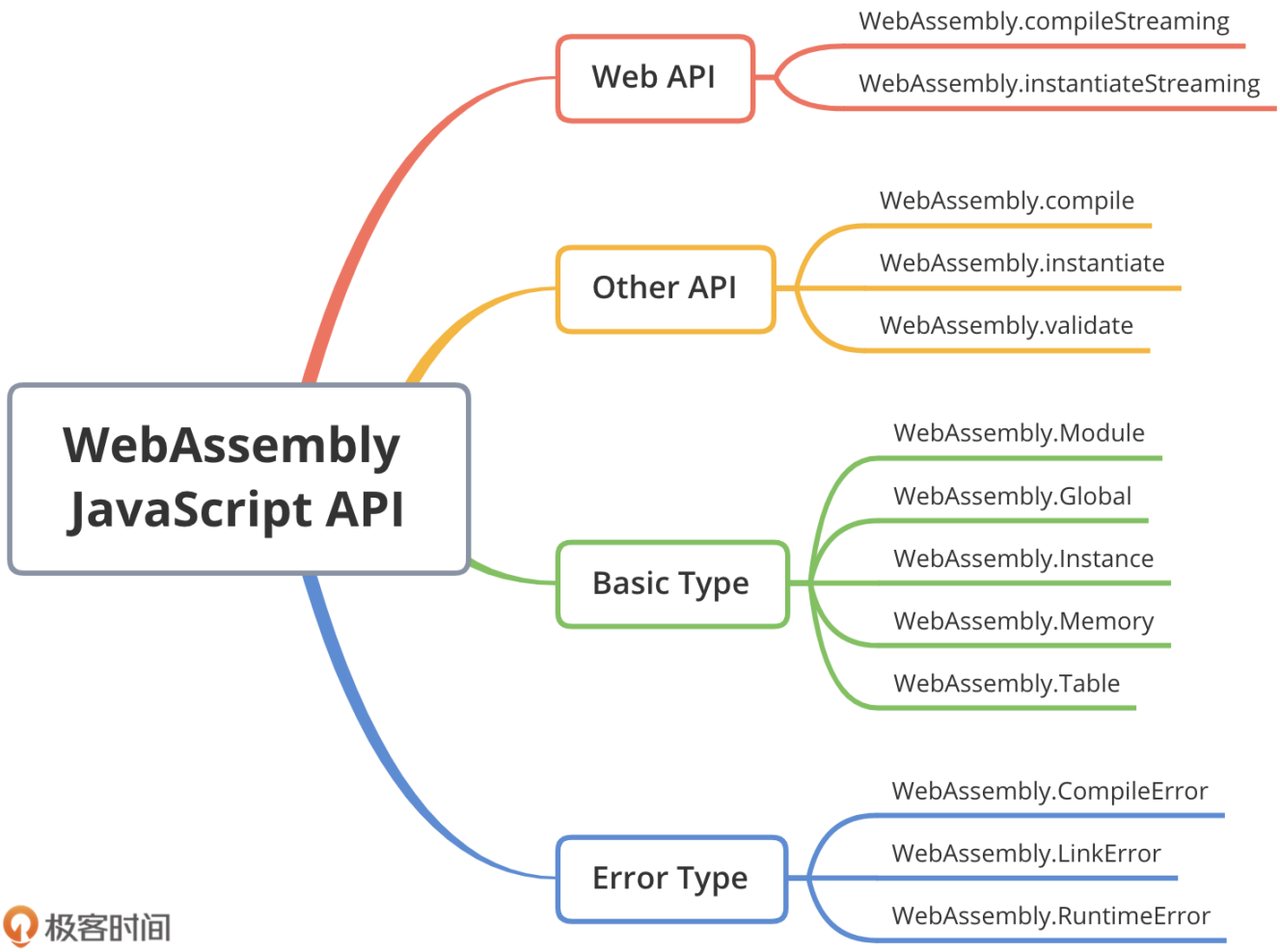
JavaScript API 提供了众多的包装类型，这样便能够在 JavaScript 环境中表示 Wasm 模块的不同组成部分。比如 WebAssembly.Moulde 对应的 Wasm 模块对象、WebAssembly.Memory 对应的 Wasm 线性内存对象等等。

除此之外，JavaScript API 中还提供了诸如 WebAssembly.Compile 以及 WebAssembly.instantiate 方法，以用于编译及实例化一个 Wasm 模块对象。

相对的，Web API 则提供了与 Web 平台相关的一些特殊方法。比如 WebAssembly.compileStreaming 与 WebAssembly.instantiateStreaming。借助这两个 API，我们可以更加高效地完成对 Wasm 模块对象的编译和实例化过程。

除此之外，我们还讲解了 Wasm 模块在运行时的一些特征，比如“内存模型”。以及目前在 MVP 标准下应用 Wasm 时的一些局限性等等。相信学完本次课程，你可以对 “Wasm 目前在 Web 平台上能够做些什么，哪些事情暂时还无法做到？” 这个问题，有着一个更加深刻的认识。

最后，我绘制一个 Wasm JavaScript API 脑图，可以供你参考以及回顾本节课的内容。



课后思考

最后，我们来做一个思考题吧。

如果你是曾经使用过 Wasm 的同学，那么你觉得在目前的 MVP 标准下，Wasm 还有着哪些局限性亟待解决？如果你还没有使用过 Wasm，那么你最期待 Wasm 能够支持哪些新的特性呢？

今天的课程就结束了，希望可以帮助到你，也希望你在下方的留言区和我参与讨论，同时欢迎你把这节课分享给你的朋友或者同事，一起交流一下。

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 07 | WASI：你听说过 WebAssembly 操作系统接口吗?

下一篇 09 | WebAssembly 能够为 Web 前端框架赋能吗?

精选留言 (6)

写留言

空间
2020-10-01

目前局限性（不是很确定）：
1. 不能脱离JS环境
2. 和JS的相互调用又不好用，麻烦
3. 虚拟机没有像JS一样访问浏览器功能的诸多接口如DOM, webgl, 定位，传感器，语音...
这些可以作为虚拟机在浏览器上提供的基础库，如果是其他环境（如作为云服务网格的...

展开

1

大土豆
2020-09-21

获取完之后，没有全部编译成平台相关的机器码吧？还有一部分字节码，解释执行

作者回复: 嗯嗯，我这里讲的可能不太严谨，其实应该是平台相关的代码，比如V8，那就是V8内部用于表示WebAssembly.Module 这个对象的代码组织形式。一般来说会先解释执行字节码来快速启动，然后再由JIT编译成机器码。当然具体实现可能有所不同。

1

慌慌张张
2020-10-23

您好老师，麻烦问一下wasm在浏览器中执行的时候，也有内存对齐这一说吗？

作者回复: 有的哈，诸如 i32.store 等指令对应的立即数都是由 offset 和 alignment 两部分组成的。

军秋
2020-09-27

wasm和video若能更好的结合，解决播放H265的视频就好了

一步
2020-09-24

在 nodejs 的环境中进行测试，也是有上面那些 WebAssembly 的构造函数和方法的，那么可以直接在 nodejs 的生产环境中使用吗？ 兼容程度是怎么样的?

展开

作者回复: 关于兼容性可以在 MDN 上对应各个 Wasm API 最下面的兼容性列表中查看哈。目前来看，在最新版的 Nodejs 中，除了 Streaming 相关的 API，其他的应该都有支持的，因此是可以直接使用的。而关于对 WASI 的兼容性，可以参考这里：https://nodejs.org/api/wasi.html

青史成灰
2020-09-21

从第一课到现在，都是文本。。。多搞点代码，是不是更好些。毕竟程序员，代码比文本更加直观

作者回复: 代码会放到实战篇哈。